MATE: The Multi-Agent Test Environment - Functional Description

Cindy L. Mason
Artificial Intelligence Research Branch
Mail Stop 269-2
NASA Ames Research Center
Moffett Field, CA 94035-1000

NASA Ames Research Center
Artificial Intelligence Research Branch

Technical Report FIA-93-09

June, 1993

MATE: The Multi-Agent Test Environment - Functional Description

Cindy L. Mason

AI Research Branch, Mail Stop: 269-2
NASA Ames Research Center
Moffett Field, CA 94035 U.S.A.

mason@chaos.arc.nasa.gov

Abstract

In this report we present the Multi-Agent Test Environment, MATE. MATE is a collection of experiment management tools for assisting in the design, testing, and evaluation of distributed problem-solvers. It provides the experimenter with an automated tool for executing and monitoring experiments choosing among rule bases, number of agents, communication strategies, and inference engines. Using MATE the experimenter can run a series of distributed problem-solving experiments without human intervention. This report is similar in content to the previous NASA Technical Report FIA-92-25 but contains an appendix detailing a number of MATE functions.

MATE: The Multi-Agent Test Environment - Functional Description

Cindy L. Mason

1 Introduction

The MATE (Multi-Agent Test Environment) testbed is a collection of experiment management tools for the design, testing, and evaluation of distributed problem-solving experiments. Using MATE the experimenter can run a series of distributed problem-solving experiments without human intervention. MATE tools use several UNIX-based workstations networked together via ethernet in a large local-area network. Each workstation runs a single agent that communicates via the local-area network, except for one workstation that serves as the MATE control center and communication server (see figure 1). Using MATE, the experimenter creates a problem-solving setting by choosing the rule bases, number of agents, inference engine, and communication strategy. Thus, MATE provides a platform for an experimenter to investigate alternative reasoning, communication and control strategies.

The role of MATE in solving a particular problem-solving instance is depicted in figure 2. The multi-agent software can be viewed in terms of 3 layers, the agent,

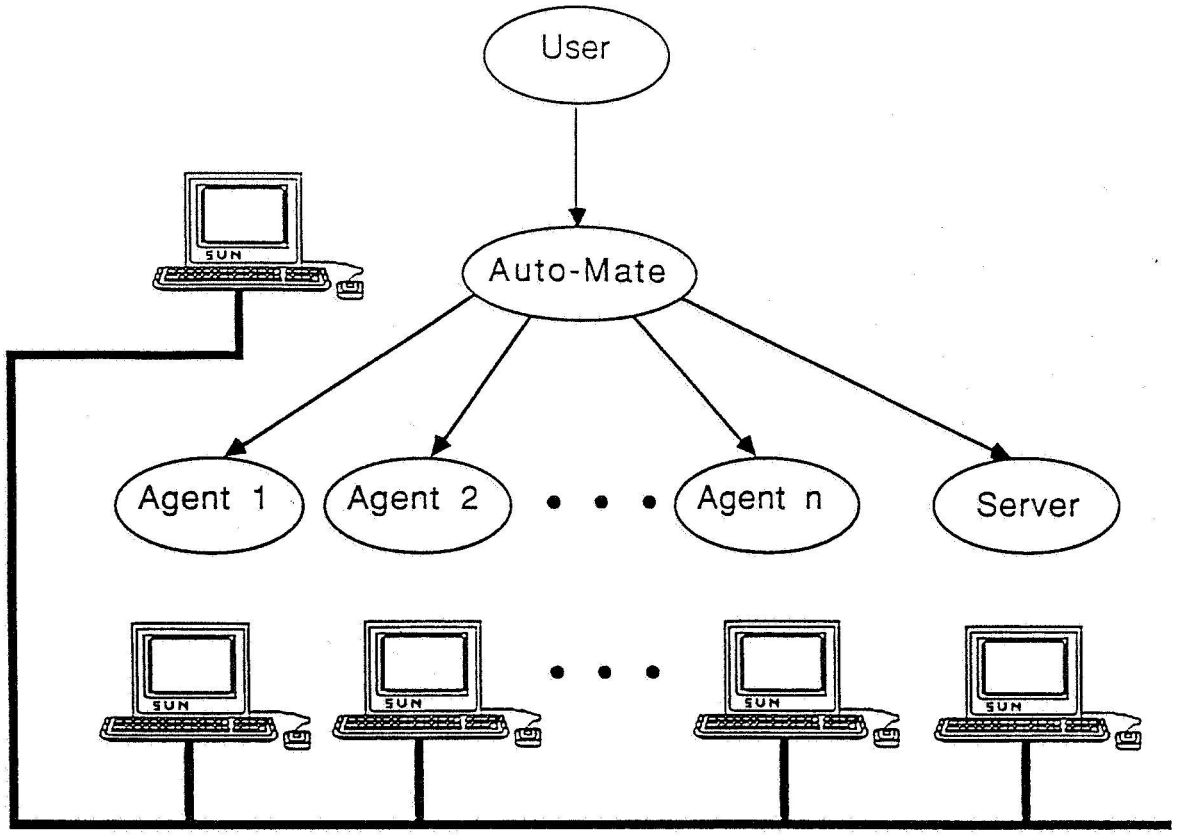

Figure 1: The MATE testbed architecture consists of the control and monitoring program, AUTO-MATE, the communications server, and one or more agents, interconnected by an ethernet.

the communications layer, and MATE. An "agent" is composed of a lisp interpreter, inference engine, rule base, data set, and a communications interface.

The communication layer provides four services to the agents: 1) point-to-point, multi-cast, and broadcast message delivery, 2) transparent translation between symbolic agent name and physical workstation address, 3) experiment start synchronization, and 4) experiment termination notification.

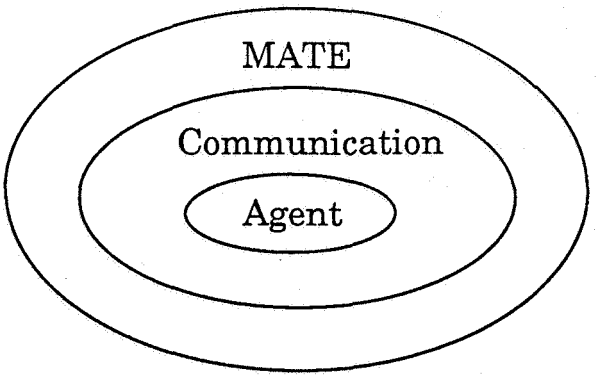

Figure 2: An agent consists of the individual Agent layer, a Communications layer, and the MATE control layer.

2 MATE Tools

MATE tools automate much of the work involved in monitoring and organizing the execution of the distributed problem solver. MATE's primary control tool is "AUTO-MATE" which finds a set of eligible machines and executes preconfigured experimental trials. Alternatively, experiments may be run manually via the program "LAUNCH-MATE", which may be selected from a menu in the X-windows user interface (see figure 3), or simply typing the program name after the UNIX command prompt.

Using AUTO-MATE, each experimental trial is automatically monitored for network, machine, or software failure and is restarted if necessary and possible. The results may be reviewed at the time the experiments are run or much later, as each trial log is automatically archived for later analysis. These logs are used not only for gathering performance information but in debugging the distributed problem solver as well.

Figure 3: The Manual MATE User Interface

There are three phases to running experiments: 1) Experiment configuration, 2) network resource allocation, and 3) experiment execution and recording. Each of these phases is described below.

2.1 Experimental Configuration

Each experiment must be configured according to number of agents (and therefore the number of machines to be used), communication policy, inference engine, rule bases and data sets. The configuration of experiments is done manually, by specifying a collection of invocation parameters to MATE tools. Currently, communication policies may be selected as broadcast, undirected-request, and directed-request. Three inference engines may be selected: 1) **single** - for debugging a single agent with no communication 2) **non-tms** - for debugging multiple agents without truth maintenance systems 3) **full-engine** - for running multiple agents with truth maintenance systems. Any number of rule bases and data sets may be created and specified for an experimental trial. In addition, new inference engines and communication policies may be added to the system by simply building the new software and appending the list of engine types (or communication policies) in the appropriate MATE tools.

2.2 Network Resource Allocation

In the network resource allocation phase, workstations in the local area network are selected for the experiments by the program, "NOSE". NOSE is called either from AUTO-MATE or run manually. NOSE consults a list of known workstations on the local area network, and selects a machine for each agent. NOSE tests each workstation to determine if it: 1)has the proper file system and account access to run the experiments,[1] 2) has the memory resources to run the lisp interpreter, 3) has sufficient CPU power and 4) the system is not heavily loaded by other users (the load threshold is adjustable.) NOSE produces a list of suitable workstations once per experimental session or whenever a negative change in usage patterns of the workstations is detected during the runs. This was found to be quite sufficient, and much less expensive than testing for workstation availability before each trial.

2.3 Experiment Execution and Recording

Once the experiment configuration is in place, and a list of capable machines is created, the distributed problem solver may be initiated through the AUTO-MATE control and monitoring program (implemented as a UNIX csh program), or manually via menu selections (see figure 5.3). Typically, manual invocation of the system is

[1] In this implementation we use NFS, the UNIX-based network file system, for setting up, controlling, and recording the experiments. Other implementation choices are possible and will not affect the experimental results. The agents communicate between themselves via the communication server which is independent of NFS.

performed while debugging new rule sets, inference engines, etc., as the experimenter is given finer control over execution and can interact with the lisp interpreter, run single step, and watch rule tracings and assumptions data base creation, etc.

The main purpose of AUTO-MATE is to execute a problem-solving system over a variety of problem scenarios, making sure everything goes smoothly. This includes:

- ensuring agent logs are written properly
- ensuring that all agents are up and running - restarting agents whenever error conditions allow
- verifying that no agent terminates prematurely
- detecting when an experiment is over
- detecting error conditions

For each experimental trial, AUTO-MATE begins by copying the proper inference engine initialization file and rule-sets (specified by invocation parameters) for this particular trial into place. AUTO-MATE then runs the communication server, watching to ensure it starts successfully[2]. Should the server have any errors, AUTO-MATE will persistently re-start the server until it runs successfully. At this point each agent is then initiated on the remote workstations using the UNIX remote shell execution command, rsh. As a result of the rsh, a UNIX csh script is executed on each machine that, after killing off any other agents found on the machine, will start

[2]The communications server is needed for instrumentating the experiment, and will not be necessary in a deployed system. However, agents must either address the problem of how to synchronize problem solving, or have the ability to maintain results on multiple problem instances.

up common lisp, thus loading in and executing the agent process. This part of the experiment system is subject to several types of failures (e.g. a machine has crashed, the network went down, there is not enough memory available to start lisp) so AUTO-MATE closely monitors log files for the agents. If necessary, AUTO-MATE will try to restart one or more agents that did not start, and may re-run NOSE looking for other available machines.

The UNIX csh script invoked on each workstation starts the process to build and execute an agent. The remote shell script first outputs a recognizable welcome message to the log file. This informs the AUTO-MATE monitoring process that agent initialization has begun successfully. After the message is output, the remote script invokes the lisp interpreter and reads the initialization file, loading the inference engine and rule and data-set, and begins execution of the remote agent. At this point the lisp agent makes a connection with the communication server and waits for a synchronization message to start processing inferences. This synchronization message signals that all agents are ready to begin. During problem solving, agents write many intermediate results and runtime statistics (such as size of working memory, number of messages sent or received, CPU times, and so on) to the log file as each step of the processing proceeds. When the agent can make no further inferences with its current data base, it enters a loop and waits for any messages that could trigger further inferences. A recognizable message is written to the log file when this loop is entered.

Since the communication server must be prepared to process broadcast messages, and the names and addresses of the agents are not known to the server until a connection is made, all agents are required to connect with the server before any messages are delivered. The communications server establishes port connections for agents, using streams, and waits for a connection message from each agent. As connections are made, the server builds up its name translations table. Once all connections are made, the server sends a "Gentlemen start your inference engines" synchronization message to all agents. This synchronization procedure prevents fast agents from queueing up large numbers of incoming messages in the server before they can be delivered to the agents. Without this synchronization, a single slow agent will delay the delivery of messages to all agents.

When all agents are finished processing and are waiting for further messages, the AUTO-MATE monitoring process detects that the trial is complete (by recognizing that this pattern is at the end of each log file for a fixed time period) and terminates the communication server and the agents. The log files for each of the agents and the communication server are copied into the proper place in the directory structure for that experimental trial, and the next trial is started.

3 Summary

The MATE (Multi-Agent Test Environment) testbed is a collection of experiment management tools for the design, testing, and evaluation of distributed problem-solving experiments. These tools use several UNIX-based workstations networked together via ethernet in a large local-area network. MATE can be used to run distributed experiments automatically and monitors experiment execution and completion.

Appendix A

Auto-Mate Functions

This appendix contains documentation for a number of functions developed for the automated control of experimentation in MATE. These functions occur in shell files, but may also be typed at the shell interpreter as individual commands.

auto-mate n-agents dir time-limit > & log-file

n-agents	The number of agents in this run of experiments
dir	The directory containing the rule base sets for the experimental trials
time-limit	If the experiments are not completed by this time, stop, where time is mmddhh.
log-file	The name of a file to which all output from auto-mate is recorded.

Example: auto-mate 15 trd 050707 > & trd.log1

Auto-mate is the top level command entered by the experimenter. Auto-mate is a csh script which runs a sequence of experimental trials using the rules and inference engine specified by the "dir" parameter and the number of agents specified by the "n-agents" parameter. The particular sequence of network configurations is specified in the auto-mate script file itself (and can be modified by editing the script file). Auto-mate first copies the proper inference engine load file to the default common lisp load file, ".clinit.cl", and invokes auto-link to prepare the rule sets and result directories for each network configuration. Auto-mate then invokes auto-mate-run-scene once for each experimental trial to actually run the experiments. The file specified by the parameter "log-file" will contain a record of the entire set of experiments.

auto-mate-kill-wait

Example: auto-mate-kill-wait

Auto-mate-kill-wait makes sure everything has died from previous runs when auto-mate is first started. It first checks for the presence of "auto-mate-run-scene" or "auto-mate-launch-processes" in the PS table. If either exist, auto-mate exits. Otherwise auto-mate-kill-wait waits for any processes, whose task is to kill auto-mate processes, to finish, before allowing auto-mate to continue. This program is run as a subprocess of auto-mate.

auto-mate-killer

Example: auto-mate-killer

Auto-mate-killer's purpose is to terminate an experimental trial. This is accomplished by killing the communications server. Auto-mate-killer is invoked when either the experiment has finished or when the experiment must be restarted (by auto-mate-run-scene or auto-mate-launch-processes). If by some chance an agent does not die when the communications server dies, auto-mate-killer will also kill this process. This is accomplished by scanning the output of the UNIX "ps" command output for the existence of an auto-mate-remote-agent process.

auto-mate-launch-processes n-agents time-limit

n-agents	The number of agents in this run of experiments
time-limit	If the experiments are not completed by this time, stop, where time is mmddhh.

Example: auto-mate-launch-processes 15 050707

auto-mate-launch-processes is a csh script that fires up the communication server and the agent processes on the remote workstations and monitors them to ensure they started properly, stopping and retrying if necessary.

Auto-mate-launch-processes first tests that the number of hosts in the file "auto-mate-hosts" is greater than or equal to the number specified by the "n-agents" parameter. If not then a new "auto-mate-hosts" file is generated by running "nose".

Auto-mate-launch-processes then fires up the communication server and monitors that it started properly. If an error is detected the server is killed, a wait is performed to allow the network operating system software to reset, and the server is restarted.

Once the server is up, auto-mate-launch-processes then fires up each agent on a remote workstation with a "rsh host auto-mate-remote-agent" command, using

the workstations specified in the "auto-mate-hosts" file. Auto-mate-launch-processes then monitors the log files of the agents until it determines that they have started properly or an error is detected. If the "rsh" command fails, it can be retried without stoping the other agents or server. If the "rsh" command succeeds but the "auto-mate-remote-agent" command fails on the remote host then the entire experiment, server and agents, is termintated and started over.

auto-mate-remote-agent agent-number max-agents

agent-number	The index of the agent to run
max-agents	The total number of agents in this experiment

Example: auto-mate-remote-agent 10 15

Auto-mate-remote-agent is a csh script that runs on the remote workstations. This script cleans up any previous files in /tmp, outputs a welcome message to the script file (so it can be detected by auto-mate-launch-processes), outputs some identifing info to the script file for debugging purposes, invokes common lisp to actually run the agent redirecting output to the script file, and outputs exit message to script file after lisp exits.

auto-mate-run-scene dir max-agents time-limit min-agents

dir	The directory containing the rule sets for an experimental trial
max-agents	The maximum number of agents in this run of experiments
time-limit	If the experiments are not completed by this time, stop, where time is expressed as mmddhh
min-agents	The minimum number of agents in this run of experiments

Example: auto-mate-run-scene trd/scene2.1/scene2.1.3 15 050707 15

Auto-mate-run-scene is a csh script which runs a single experimental trial, or a sequence of trials using the same rules, varying the number of agents. Auto-mate-run-scene first checks the time limit, checks for the existence of the "auto-mate-abort" file, copies the rules specified by the "dir" parameter to a default load directory, creates a directory to hold the log files for each number of agents, and deletes any left over agent or server log files. Auto-mate-run-scene then invokes auto-mate-launch-processes to fire up the server and the agents on remote workstations.

Auto-mate-run-scene then monitors the agent log files to determine when the experimental trial is complete, or if an error has occured. The agent log files are

tested by running auto-mate-script-tester. If an error is detected, the experiment is terminated and restarted from the beginning. When the experiment is detected as being completed, the server is terminted with a "kill -9" command. Each agent connected to the server will detect the server's death and terminate itself. Auto-mate-run-scene will then copy all the agent log files their final directories.

auto-mate-script-tester agent-number

agent-number the index of the agent whose script is to be tested

Example: auto-mate-script-tester 6

Auto-mate-script-tester is a csh script that examines the log file of an agent (the output of auto-mate-remote-agent) to determine the state of the agent. There are three states an agent may be in: 1) it has encountered an error 2) it is waiting for a message 3) it is proceeding with problem-solving. States 1 and 3 cause auto-mate-script-tester to create files to indicate the agent is either in error and needs to be restarted, or that the agent is still working.

If there has been an error, auto-mate-script-tester will detect the "Error: ..." message in the log file. Auto-mate-script-tester then removes the host machine on which the error occurred from the auto-mate-host file, and creates the auto-mate-abort-and-retry which is a flag to auto-mate-run-scene to terminate the experiment and start over. If the agent is waiting for a message, auto-mate-script-tester will detect the presence of the profile information, as it is written to the log file each time the agent goes into the wait for message loop. If neither of these states exist, then it must be the case that the agent is still working. In this case auto-mate creates the auto-mate-still-thinking file, which is a flag to the auto-mate- run-scene to continue monitoring the agents.

auto-link number-of-agents

number-of-agents the number of agents whose rules sets should be symbolically linked

Example: auto-link 15

Auto-link makes symbolic links for all the experiment rule bases

kill-everything

Example: kill-everything

Kill-everything kills any processes which have "auto-mate" as part of the process name. The purpose of the program is to stop any processes which are still executing."

remote-kill-all

Example: remote-kill-all

Remote-kill-all's purpose is to go out over the net and detect and kill any zombie lisp processes which may still be hanging around. This is accomplished by examining the auto-mate-host table and executing a "remote-killer" processes on each machine found in the auto-mate-hosts table.

remote-killer

Example: remote-killer

Remote-killer scans the processes on a machine for the existence of an auto-mate lisp process and kills it.

nose number-of-agents

number-of-agents the number of agents for whom we must find host machines

Example: nose 15

Nose examines a list of workstations to see if they are suitable for running an agent. The list is a comprehensive set of machines which are on the local area network, run UNIX, and on which we have an account. The criteria for suitability includes:

1. having the correct disk mounted via NFS (for access to executables and agents)
2. the machine architecture is a Sun4.
3. having a low load average (ie. mostly idle)
4. the resources to run common lisp (sufficent processes, memory, swap space etc.)

Nose also examines the machines for left over lisp processes and kills them if they exist. To find the suitable workstations, NOSE invokes the subprocess nose-try for each machine in the list. For each suitable machine, an entry is made in the file "auto-mate-hosts."

nose-try host-name number-of-agents

host-name	The host to test for the ability to run an agent
number-of-agents	The number of hosts we need to find

Example: nose-try zaphod.es.llnl.gov 15

Nose-try is a csh script which tests a single host for the ability to run and agent remotely. Nose-try first tests the length of the auto-mate-hosts list against the parameter "number-of-agents" and terminates if enough hosts are already found. If more hosts are needed nose-try launches a "rsh host-name remote-nose" command to the remote host to test that host.

remote-nose

Example: remote-nose

Remote-nose is a csh script which runs on the remote workstation that is being tested for the ability to run an agent. Remote-nose tests the host for the criteria described under "nose". If these criteria are met then remote-nose appends this host name to the list auto-mate-hosts.

server n-agents time-limit log-file

n-agents	the number of agents for the server to connect
time-limit	If the experiments are not completed by this time, stop.
log-file	The name of a file containing the record of all communication and server messages

Example: server 15 041512 server15.log

Server is a "C" program which provides communication services between agents and gathers runtime statistics.